PATENT
5150-65801

"EXPRESS MAIL" MAILING
LABEL NUMBER EL990144380US
DATE OF DEPOSIT MARCH 15, 2004
I HEREBY CERTIFY THAT THIS PAPER OR FEE IS BEING DEPOSITED WITH THE UNITED STATES POSTAL SERVICE "EXPRESS MAIL POST OFFICE TO ADDRESSEE" SERVICE UNDER 37 C.F.R. § 1.10 ON THE DATE INDICATED ABOVE AND IS ADDRESSED TO THE COMMISSIONER OF PATENTS AND TRADEMARKS, ALEXANDRIA, VA 22313-1450

Derrick Brown

RUN-TIME NODE PREFETCH PREDICTION IN DATAFLOW GRAPHS

By:

Newton G. Petersen

Attorney Docket No.: 5150-65801

Jeffrey C. Hood/MRW
Meyertons, Hood, Kivlin, Kowert & Goetzel PC
P.O. Box 398
Austin, Texas 78767-0398
Ph: (512) 853-8800

Priority Claim

This application claims benefit of priority of U.S. provisional application Serial No. 60/454,771 titled "Node Prefetch Prediction in Dataflow Graphs" filed March 14, 2003, whose inventor is Newton G. Petersen.

Field of the Invention

The present invention relates to optimizing execution of dataflow programs, and particularly to run-time prediction and prefetching of state information for nodes in a dataflow program.

Description of the Related Art

Statically schedulable models of computation do not need prediction because all scheduling decisions are made at compile time; however, they can still take advantage of prefetching state data. Synchronous dataflow (SDF), computation graphs, and cyclo-static dataflow (CSDF) are all powerful models of computation for applications where the schedule is known at compile time. For a valid schedule, it is possible to speed-up the process by simply pre-loading actors and their respective internal state data.

Figure 1 (Prior Art)

Figure 1 shows the prefetch nodes explicitly in the diagram. However, the prefetch nodes could be added implicitly when targeting hardware capable of taking advantage of the parallelism exposed in the dataflow graph.

The idea of prefetching data is not new. Cahoon and McKinley have researched extracting dataflow dependencies from Java applications for the purpose of prefetching state data associated with nodes. Wang et al., when exploring how to best schedule loops expressed as dataflow graphs, also try to schedule the prefetching of data needed for loop iterations.

Figure 2 (Prior Art)

Figure 2 illustrates a homogeneous and quasi-static Boolean Data Flow graph, according to Prior Art. While statically schedulable models of computation are common in signal and image processing applications and make efficient scheduling easier, the range of applications is restrictive because runtime scheduling is not allowed. Dynamically schedulable models of computation, such as Boolean dataflow, dynamic dataflow, and process networks, allow runtime decisions, but in the process make static prefetch difficult, if not impossible. As J. T. Buck explains in his thesis, Boolean Dataflow (BDF) is a model of computation sometimes requiring dynamic scheduling. The *switch* and *select* actors allow conditional dataflow statements with semantics for control flow as shown in Figure 2.

Cyclo-dynamic dataflow, or CDDF, as defined by Wauters et al., extends cyclo-static dataflow to allow run-time, data-dependent decisions. Similar to a CSDF, each CDDF actor may execute as a sequence of phases *f1, f2, f3, f4, ..., fn*. During each phase *fn*, a specific code segment may be executed. In other words, the number of tokens produced and consumed by an actor can vary between firings as long as the variations can be expressed in a periodic pattern. CDDF allows run-time decisions for determining the code segment corresponding to a given phase, the token transfer for a given code segment, and the length of the phase firing sequence. The same arguments for using caller prediction for BDF graphs can be extended to include CDDF.

Dynamic dataflow, or DDF, extends the BDF model to allow any Boolean expression to serve as the firing rule. For example, the switch node in Figure 2 could allow a variable number of tokens on one of its inputs before firing. DDF also allows recursion. DDF requires a run-time scheduler to determine when an actor becomes executable, and is a possible model of computation that could benefit from our prefetch prediction method.

Lee and Parks describe dataflow process networks as a special case of Kahn process networks. The dataflow process networks implement multiple concurrent processes by using unidirectional FIFO channels for inter-process communication, with non-blocking writes to each channel, and blocking reads from each channel. In the Kahn

and MacQueen representation, run-time configuration of the network is allowed. While some scheduling can be done at compile time, for some applications most actor firings must be scheduled dynamically at run-time.

Figure 3 (Prior Art)

Many modern pipelined processors use branch prediction to improve performance by allowing a processor's pipelines to remain full for a greater percentage of the time and by reducing the number of times the pipelines must be flushed because the processor does not know where program execution will be after an instruction. If a processor doesn't know what the next instruction will be, it can't start decoding and executing it. Many methods of branch prediction exist to help the processor predict what the next instruction will be.

One method of branch prediction uses a saturating 2 bit counter that increments whenever a branch is taken and decrements whenever a branch is not taken. The most significant bit of the counter then becomes the prediction of whether a branch will be taken or not.

Figure 4: (Prior Art)

Figure 4 illustrates two-level branch prediction. A '1' in the counter MSB predicts that a branch will be taken, while a '0' predicts that the branch will not be taken. This approach may achieve a prediction success rate in the 90% range. Patt and Yeh have tabulated the hardware costs to be significant for large history lengths.

Two-level branch prediction was pioneered by Patt and Yeh to help keep processor pipelines full for a greater percentage of the time. This prediction model uses a lookup table of saturating two-bit counters that represent the likelihood that a branch will be taken given a certain history. As illustrated below in Figure 4, the history register consists of data indicating if a branch was taken, or not taken, the past n times. A '1' represents a taken branch, and a '0' represents a branch not taken. The table therefore has 2^n entries.

It would be beneficial to have a mechanism for run-time optimization of a dataflow program that includes dynamic constructs.

Summary

A method and system for run-time prediction of a next caller of a shared functional unit for a data flow program is presented. For example, in a multiprocessor system, functional units of an application require different state information depending on the caller of the functional unit. Examples of such functional units include control and filtering applications, among others.

For example, control applications such as PID loops may require saving a last error and a running integral value as state information associated with a particular caller. A filtering application may require saving previous output values for an IIR filter and/or previous input values for FIR and/or IIR filters. Additionally, different callers may require different filter coefficients. These coefficients could be considered additional state information.

State information is usually saved in memory and may be fetched sequentially every time the functional unit is run. In a multiprocessor system with separate memory banks and caches, the processor running the functional unit may have idle time while other parts of the application are running on different processors. During this dead time, the processor could be fetching the state information from memory into it's cache assuming it knows which state information it needs. It could know this by using call prediction and would save all the time necessary to fetch the state information from memory. This could result in a significant speedup. Similar speed up could be achieved for other hardware units capable of parallel execution.

A method for run-time prediction of a next caller of a shared functional unit, wherein the shared functional unit is operable to be called by two or more callers out of a plurality of callers would enable prefetching of state information during the 'dead time'. The shared functional unit and the plurality of callers are operable to execute in parallel on a parallel execution unit. The run-time prediction is used for data flow programs. The run-time prediction can detect a calling pattern of the plurality of callers of the shared functional unit and predict the next caller out of the plurality of callers of the shared functional unit. The run-time prediction then loads state information associated with the next caller out of the plurality of callers.

The shared functional unit and the plurality of callers are operable to execute in parallel on a parallel execution unit. The parallel execution unit includes an FPGA, a programmable hardware element, a reconfigurable logic unit, a nonconfigurable hardware element, an ASIC, a computer comprising a plurality of processors, and any other computing device capable of executing multiple threads in parallel. In other words, the run-time caller prediction may be implemented on a variety of the parallel execution units that execute data flow programs, such as a multi-processor computing device capable of parallel execution, configurable hardware elements such as FPGAs, and non-configurable hardware elements such as ASICs. The application that executes on the parallel execution device may be programmed in a data flow language such as LabVIEW.

Brief Description of the Drawings

A better understanding of the present invention can be obtained when the following detailed description of the preferred embodiment is considered in conjunction with the following drawings, in which:

Figure 1 illustrates a homogeneous synchronous data flow graph that uses prefetching of state information for a shared node D, according to Prior Art;

Figure 2 illustrates a homogeneous and quasi-static Boolean Data Flow graph, according to Prior Art;

Figure 3 illustrates a saturating two-bit branch prediction mechanism, according to Prior Art;

Figure 4 illustrates a two level branch prediction mechanism, according to Prior Art;

Figure 5 is a block diagram of a run-time caller prediction mechanism, according to one embodiment;

Figure 6 is a flowchart diagram illustrating a run-time caller prediction, according to one embodiment;

Figure 7 illustrates a two level caller prediction mechanism, according to one embodiment;

Figure 8 illustrates a periodic caller prediction mechanism, according to one embodiment;

Figure 9 illustrates exemplary LabVIEW VIs for testing an exemplary periodic caller prediction, according to one embodiment;

Figure 10 is a table of exemplary execution times of the exemplary LabVIEW VIs without using the periodic caller prediction, according to one embodiment; and

Figure 11 is a table of exemplary execution times of the exemplary LabVIEW VIs using the periodic caller prediction, according to one embodiment.

While the invention is susceptible to various modifications and alternative forms specific embodiments are shown by way of example in the drawings and will herein be described in detail. It should be understood however, that drawings and detailed description thereto are not intended to limit the invention to the particular form disclosed. But on the contrary the invention is to cover all modifications, equivalents and alternative following within the spirit and scope of the present invention as defined by the appended claims.

Detailed Description of the Embodiments

Incorporation by Reference

The following U.S. Patents and patent applications are hereby incorporated by reference in their entirety as though fully and completely set forth herein.

U.S. Patent Application Serial No. 10/058,150, titled "Reconfigurable Measurement System Utilizing a Programmable Hardware Element and Fixed Hardware Resources" filed October 29, 2001,whose inventors are Brian Keith Odom, Joseph E. Peck, Hugo A. Andrade, Cary Paul Butler, James J. Truchard, Newton Petersen and Matthew Novacek.

U.S. Patent Application Serial No. 09/229,695 titled "Reconfigurable Test System" filed on January 13, 1999, whose inventors are Arthur Ryan, Hugo Andrade, and Brian Keith Odom, which is now U.S. Patent No. 6,322,249.

U.S. Patent Application Serial No. 06/312,242 titled "System and Method for Graphically Creating, Deploying and Executing Programs in a Distributed System" filed on August 14, 2001, whose inventors are Jeffrey L. Kodosky, Darshan Shah, and Steven W. Rogers.

U.S. Patent Application Serial No. 09/745,023 titled "System and Method for Programmatically Generating a Graphical Program in Response to Program Information," filed December 20, 2000, whose inventors are Ram Kudukoli, Robert Dye, Paul F. Austin, Lothar Wenzel and Jeffrey L. Kodosky.

U.S. Patent Application Serial No. 09/595,003 titled "System and Method for Automatically Generating a Graphical Program to Implement a Prototype", filed June 13, 2000, whose inventors are Nicolas Vazquez, Jeffrey L. Kodosky, Ram Kudukoli, Kevin L. Schultz, Dinesh Nair, and Christophe Caltagirone.

U.S. Patent Application Serial No. 09/886,455 titled "System and Method for Programmatically Generating a Graphical Program in Response to User Input", filed June 20, 2001, whose inventors are Jeffrey D. Washington, Ram Kudukoli, Robert E. Dye and Paul F. Austin.

U.S. Patent No. 4,914,568 titled "Graphical System for Modeling a Process and Associated Method," which issued on April 3, 1990, whose inventors are Jeffrey L. Kodosky, James J. Truchard, and John E. MacCrisken.

There are a few characteristics of applications that can take advantage of caller prediction model for prefetching node instance data. In one embodiment, applications should not execute in hard real-time since the prediction logic may be wrong, which may result in jitter. In one embodiment, applications for run-time caller prediction may include applications where quality of service (QoS) can vary depending on how fast the application is running. Buttazzo et al. point out that these applications are common due to the non-deterministic behavior of common low-level processor architecture components, such as caching, prefetching, and direct memory access (DMA) transfers. Buttazzo's work suggests voice sampling, image acquisition, sound generation, data compression, video playback, and certain feedback control systems as application domains that can function at varying QoS depending on the execution rate of the application.

Feedback control systems are particularly interesting because they may contain several subsystems that share common components. Abdelzaher et al. explore a complex real-time automated flight control system that negotiates QoS depending on the load of the system. The flight control system consists of a main function that controls the altitude, speed, and bearing of the plane, and also contains subsystems for flight control performance. Continuous analysis of all of the inputs to achieve optimal control is a large computation task. However, the task of flying a plane can be accomplished with relatively little computation power by carefully reserving resources and by tolerating less than optimal control functionality. PID (Proportional-Integral-Derivative) control is a possible compelling application domain because PID tuning parameters are relatively insensitive to rate changes. Many measurement and control applications may display similar properties of varying acceptable QoS.

Figure 5 – Block diagram of run-time caller prediction

Figure 5 illustrates a block diagram of the run-time caller prediction, according to one embodiment. The run-time caller prediction operates to optimize execution of nodes in a data flow program, such as a LabVIEW application described below with reference to Figure 9.

In one embodiment, the caller prediction consists of two or more callers, such as a first caller 2A, a second caller 2B, and a third caller 2C. A control and arbitration logic unit 4, such as described below with reference to Figures 6 and 7, may operate to predict a next caller of a functional unit, such as the shared functional unit 6. The shared functional unit may also be referred to as the shared functional node, such as a node in a data flow program, e.g., a LabVIEW graphical program.

The shared functional unit and the plurality of callers are operable to execute in parallel on a parallel execution unit. In one embodiment, the parallel execution unit includes an FPGA, a programmable hardware element, a reconfigurable logic unit, a nonconfigurable hardware element, an ASIC, a computer comprising a plurality of processors, and any other computing device capable of executing multiple threads in parallel. In other words, the run-time caller prediction may be implemented on a variety of the parallel execution units that execute data flow programs, such as a multi-processor computing device capable of parallel execution, configurable hardware elements such as FPGAs, and non-configurable hardware elements such as ASICs. The application that executes on the parallel execution device may be programmed in a data flow language such as LabVIEW.

For example, for an application with three callers of a shared node, all possible combinations of calls for a history length of 3 are:

0: 111
1: 112
2: 113
3: 121
4: 122
5: 123
6: 131
7: 132
8: 133

9: 211
10: 212
11: 213
12: 221
13: 222
14: 223
15: 231
16: 232
17: 233
18: 311
19: 312
20: 313
21: 321
22: 322
23: 323
24: 331
25: 332
26: 333

It is noted that Figure 5 is exemplary only, and the units may have various architectures or forms, as desired.

Figure 6 – Flowchart of run-time caller prediction algorithm

Figure 6 is a flowchart of the run-time caller prediction algorithm, according to one embodiment. The run-time caller prediction algorithm may predict a next caller of a node in a data flow program, such as a next caller of a node 32 in a LabVIEW graphical program described below with reference to Figure 9.

In 100, the control and arbitration logic may detect a calling pattern of the plurality of callers of the shared functional unit, according to one embodiment. In other words, the control and arbitration logic may examine a past history of callers of the shared functional unit 6, and detect a calling pattern. In one embodiment, the detection may operate in a manner similar to one described below with reference to Figure 7. In another embodiment, the detection may operate in a manner similar to one described below with reference to Figure 8.

In 102, the control and arbitration logic may predict a next caller of the shared functional unit 6. In other words, the control and arbitration logic may predict the next caller out of the plurality of callers, such as the next caller out of the three callers 2A, 2B, and 2C described above with reference to Figure 5. In one embodiment, the prediction may operate in a manner similar to one described below with reference to Figure 7. In another embodiment, the prediction may operate in a manner similar to one described below with reference to Figure 8.

In 104, the control and arbitration logic may load state information associated with the next caller out of the plurality of callers. In other words, the control and arbitration logic may load the state information for the next caller out of the plurality of callers, such as the next caller out of the three callers 2A, 2B, and 2C described above with reference to Figure 5. The state information may include execution state, values of any variable, any previous inputs, any previous outputs, and any other information related to execution of a node in a dataflow diagram.

In some embodiments, a memory medium may include instructions to generate a program to perform run-time call prediction of a next caller of a shared functional unit. In one embodiment, the program may operate to generate the shared functional unit and the plurality of callers from the dataflow graph on the parallel execution unit prior to detecting of 100. In other words, the units used in the data flow program as well as the prediction logic may be generated and deployed on a parallel execution unit.

The program may execute on a computer and the program may be intended for deployment on parallel execution unit. In one embodiment, the program may generate a shared functional unit and the plurality of callers from the dataflow graph on the parallel execution unit. For example, in one embodiment the program may include program instructions for a multi-processor computing device capable of parallel execution. In another embodiment, the program may include digital logic for configurable hardware

elements such as FPGAs. In yet another embodiment, the program may hardware descriptions for non-configurable hardware elements such as ASICs.

It is noted that the flowchart of Figure 6 is exemplary only. Further, various steps in the flowchart of Figure 6 may occur concurrently or in a different order than that shown, or may not be performed, as desired. Also, various additional steps may be performed as desired.

Figure 7 – Two Level Caller Prediction

Figure 7 illustrates a block diagram of a two level caller prediction mechanism, according to one embodiment. This design of the prediction mechanism may be similar to the two-level branch prediction mechanism described above with reference to Figure 4. Figure 7 shows the overall block diagram architecture. In the following example, each caller has an exemplary unique identification, i.e., is numbered from 1-9. Other unique identifications for the callers can be used as desired.

In one embodiment, a past call history register 10A, such as a shift register, may keep track of the last *n* calls to a shared node, such as the shared node 6 described above with reference to Figure 5. In one embodiment, the values held in the call history register 10A may be connected directly to a hash function 54 that may convert the history entry to a row lookup index into the call history table 11. In one embodiment, the call history table 11 may have a column for each possible caller of the shared node. In other embodiments, the call history table 11 may use a reduced and optimized numbed of columns for the callers of the shared node.

In one embodiment, each cell in the call history table 11 may be managed by update logic 56 that may operate to increment, i.e., use table update 62, when a prediction is correct and decrement when a prediction is incorrect. For example, in order to generate a prediction, the column values across the call history table 11 may be compared to find the greatest value.

In one embodiment, there may be a row in the call history table 11 for each possible value of the call history register 10A. However, the number of rows may grow

exponentially (x^n) with the length of the call history register 10A. For three callers (x) and a history length of eight (n), the direct map approach may require 6561 rows. In one embodiment, a hashing function 54A may be used to eliminate the exponential dependence, i.e., to perform a table lookup 60. In other embodiments, other algorithms may be used that minimize the number of rows in the call history table 10A.

In the above prediction model, the maximum number of callers may be analogous to a numeric base, and each position in the call history shift register 10A may be analogous to a position in the numeric base. For example, for ten possible callers, a call history register 10A containing *9,6,2* maps to a lookup index of 962 using base 10. For *11* possible callers, a conversion may be performed by evaluating *9*11^2 + 6*11^1 + 2*11^0*. In one embodiment, a hardware modulo hash function may be used to convert and multiply the running sum by a large prime number during each stage. The hardware modulo hash function may only use the lower k bits of the direct mapped result. Hashing introduces collisions, which decrease the prediction accuracy.

For the three caller example described above with reference to Figure 5, each combination may be converted into an index. For a small history length, this could be a direct conversion as shown in the example above, but for longer lengths, a hashing function may be used to keep the table sizes reasonable. For example, a history index may be used as a lookup into a table of saturating counters for each caller:

	Caller 1	Caller 2	Caller 3
0	**01**	**11**	**01**
1	**10**	**00**	**00**
2	**01**	**10**	**11**
3	**11**	**01**	**00**
...	...	...	...
15	**01**	**10**	**00**
...	...	...	...
26	**11**	**01**	**10**

Table 1: Past history lookup table for shared component callers

In one embodiment, after the table lookup, the call prediction may become the counter with the largest value. For example, for if a history of past callers is 231, then the

index may become 15 and the call prediction may predict the caller 2 as the next caller. In one embodiment, the table may be updated if the call prediction was correct.

In one embodiment, if the application quickly settles to a periodic schedule and the schedule changes infrequently, the need to keep track of past periodic behavior may be eliminated.

It is noted that Figure 7 is exemplary only, and the various elements in this Figure may have various architectures or forms, as desired. Further, various blocks in the block diagram of Figure 7 may be included in a different order than that shown, or may not be included, as desired. Also, various additional blocks may be included as desired.

Figure 8 – Periodic Caller Prediction

Figure 8 illustrates a block diagram of a run-time periodic caller prediction mechanism, according to one embodiment.

In one embodiment, Figure 8 illustrates a predictor model that determines the current periodic behavior in the current calling history. The following explanation uses call history that holds eight past callers. Note that call history size is exemplary only and other smaller and larger call history sizes can be used as desired.

In one embodiment, the periodic caller prediction may include a call history register 10 that contains a previous call history of a shared functional unit, such as the shared node described above with reference to Figure 5. In one embodiment, the call history register 10 may be updated in a manner similar to that of a shift register. In other embodiments the call history register 10 may be updated using other techniques as known in the art. In yet another embodiment, the call history may be stored in other devices such as logic cells, memory devices, and other hardware and/or software units designed to store and retrieve data.

In one embodiment, the call history register 10 in the period predictor is similar to the past history register of the two-level predictor. In one embodiment, the call history may be divided in two parts that are substantially equal to each other. In one

embodiment, comparisons 21, 22, 23, and 24 may operate to find the section of the second half of the call history that best correlates with the section of the first half. This operates to compare the callers of the first portion of the caller history to the callers of the second portion of the caller history. In one embodiment, each of the plurality of callers may have a unique identification, where the unique identification of each of the callers is used in the call history in order to identify a periodic portion of the caller history. In other embodiments other methods may be used to identify the periodic portion of the caller history.

In one embodiment, a multiplexer may be used to select an element from the call history register 10. For example, the comparisons 21, 22, 23, and 24 may operate to indicate an Equality Select 14 that may operate as an input to the multiplexer 12. The multiplexer 12 may then select an element from the call history register 10 depending on the selection input 14 from the comparisons 21, 22, 23, and 24. Other embodiments of comparing and selecting are contemplated, such as various digital logic functions and/or instructions.

For example, for a past call history of *12341234*, the first equality is true, and the periodic caller prediction may select *4* as the prediction. Similarly, for a history of *02345602*, the third equality is true, and periodic caller prediction may select *6* as the prediction. These comparisons assume that the period of the calling sequence is contained in the call history register 10, and that call prediction latency is equal to the length of the call history register 10 if the period or the calling sequence changes.

It is noted that Figure 8 is exemplary only, and the various elements in this Figure may have various architectures or forms, as desired. Further, various blocks in the block diagram of Figure 8 may be included in a different order than that shown, or may not be included, as desired. Also, various additional blocks may be included as desired.

Figure 9 – Exemplary LabVIEW filtering application

Figure 9 illustrates an exemplary LabVIEW filtering application to test an exemplary predictor model. The exemplary LabVIEW application 50 consists of two loops 40 and 42, where each loop contains an FIR filtering function. Each loop has an

ADC 30 and 36, an FIR 32A and 32B, and a DAC 34 and 28. The ADC 30 and 36 is an Analog-to-Digital Converter that is operable to acquire analog data, convert it to digital data, and supply the digital data to the FIR filter 32A and 32B. The exemplary LabVIEW application is a data flow program that may execute on any of the parallel execution units that execute data flow programs as described above with reference to Figure 5. In one embodiment, the shared functional unit and the plurality of callers are generated from a dataflow program.

The FIR 32A and 32B may be the shared node on the parallel execution unit that may be shared by two callers – the first ADC 30 in the first loop 40 and the second ADC 36 in the second loop 42. In other words, the two separate FIR nodes 32A and 32B in the LabVIEW block diagram correspond to the same shared FIR block.

The timer blocks 52 and 54 may provide the dynamic sample rate for the ADCs 36 and 38 and for the DACs 34 and 38 on the two independent channels. The sample rate may also determine which set of coefficients may be loaded into the FIR block 32A and 32B for filtering.

In one embodiment, previous inputs to the FIR filters 32A and 32B may be stored locally. The previous inputs may be dependent on the caller and thus should be saved as state information. Because each loop may be running at a different rate, the next caller of the shared node, e.g., the FIR filter node 32 may not be known. If the prediction is wrong, then state information would need to be reloaded. in other words, the worst case should not slow down the execution of the above system, and the best case speeds up the execution.

Figure 10 – Table for execution times without using Periodic Predictor

Figure 11 illustrates a table with exemplary execution times for the exemplary LabVIEW filtering application using an exemplary shared FIR block without using the periodic predictor, according to one embodiment.

In the exemplary analysis below, the first ADC 30 (block A) the second ADC 36 (block D) the first DAC (block C) and the second DAC (block E) each take 20 cycles to execute. Fetching the coefficients and current tap values (blocks F1 and F2) take 31

cycles each, and execution of the shared FIR filter block 32A and 32B (the shared node) (blocks B1 and B2) takes 6 cycles.

Figure 10 illustrates a schedule with three parallel threads of execution without prefetch prediction. In this exemplary analysis, the first loop 40 executes twice as fast as the second loop 42.

The top row of the table shows the execution time of each block, the second row shows the execution of blocks in loop 1, the third row shows when the shared FIR block executes, and the last row shows the execution of blocks in loop 2. In one embodiment, fetching may occur inline with the loop executions since it is not known which loop will call the FIR block next.

Figure 11 – Table for execution times using Periodic Predictor

Figure 11 illustrates a table with exemplary execution times for the exemplary LabVIEW filtering application using an exemplary shared FIR block using the periodic predictor, according to one embodiment.

In this example, the periodic prediction may prefetch data for the next caller while other blocks execute, as shown in Figure 9. In one embodiment, the above application may have a 33% improvement with correct predictions. In one embodiment, for a period prediction unit with a history register long enough to contain the entire period, the predictions may be correct. If the predictions are incorrect, the sampling schedule may shift, and the sampling rate would vary. Note that these results are exemplary only for a simple exemplary application and may vary depending on the implementation of the run-time prediction mechanism, the type of a data flow program, the number and size of shared nodes, and other factors as described above.

Although the system and method of the present invention has been described in connection with the preferred embodiment, it is not intended to be limited to the specific form set forth herein, but on the contrary, it is intended to cover such alternatives, modifications, and equivalents, as can be reasonably included within the spirit and scope of the invention as defined by the appended claims.